# QL Assembly Language Mailing List

Issue 3

Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

http://qdosmsq.dunbar-it.co.uk/downloads/AssemblyLanguage/Issue_003/Assembly_Language_003.pdf



This pdf document was created on *D:20160209102847Z.*

# Contents

# List of Tables

# List of Figures

# Listings

# 1. Preface

## 1.1 Feedback

Please send all feedback to `assembly@qdosmsq.dunbar-it.co.uk`. You may also send articles to this address, however, please note that anything sent to this email address may be used in a future issue of the eMagazine. Please mark your email clearly if you do not wish this to happen.

This eMagazine is created in LaTeXsource format, aka plain text with a few formatting commands thrown in for good measure, so I can cope with almost any format you might want to send me. As long as I can get plain text out of it, I can convert it to a suitable source format with reasonable ease.

I use a Linux system to generate this eMagazine so I can read most, if not all, Word or MS Office documents, Quill, Plain text, email etc formats. Text87 might be a problem though!

## 1.2 Subscribing to The Mailing List

This eMagazine is available by subscribing to the mailing list. You do this by sending your favourite browser to `http://qdosmsq.dunbar-it.co.uk/mailinglist` and clicking on the link “Subscribe to our Newsletters”.

On the next screen, you are invited to enter your email address *twice*, and your name. If you wish to receive emails from the mailing list in HTML format then tick the box that offers you that option. Click the Subscribe button.

An email will be sent to you with a link that you must click on to confirm your subscription. Once done, that is all you need to do. The rest is up to me!

## 1.3 Contacting The Mailing List

I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

I suspect George will continue to keep me correct on matters where I get stuff completely wrong, as before, and I know George did ask if the list would be contactable, so I've set up an email address for the list, so that you can make comments etc as you wish. The email address is:

`assembly@qdosmsq.dunbar-it.co.uk`

Any emails sent there will eventually find me. Please note, anything sent to that email address will be considered for publication, so I would appreciate your name at the very least if you intend to send something. If you do not wish your email to be considered for publication, please mark it clearly as such, thanks. I look forward to hearing from you all, from time to time.

If you do have an article to contribute, I'll happily accept it in almost any format - email, text, Word, Libre/Open Office odt, Quill, PC Quill, etc etc. Ideally, a LaTeXsource document is the best format, because I can simply include those directly, but I doubt I'll be getting many of those! But not to worry, if you have something, I'll hopefully manage to include it.

# 2. Bubble Sorts

Part of a little program that I'm working on requires the characters of a word to be sorted into order, ascending in this case, and as there's no trap or vector in QDOSMSQ to allow this to be easily done, I've had to work out my own. The bubble sort is one of the simplest sorting algorithms that there is, however, it is pretty inefficient as much of the work it does is checking over data that it has already sorted in any previous pass. Also, the more data there are to sort, the longer it takes to sort. Much longer in fact.

Looking on Wikipedia for some slightly improved versions, I found the one below. It doesn't reduce the number of swaps that take place, but it does 'know' that when it has made a pass through the array of bytes, in this case, the last item that it swapped is the lowest possible value for this pass, and anything from that point on in the array is already sorted. By 'knowing' it does at least reduce the number of comparisons that have to be made on each pass, which reduces the run time of the sort.

The data is sorted by moving the higher values - in this version - down the array, one place at a time, until the array's bottom end contains all the sorted data, while the top end contains the data that are yet to be sorted. Hopefully, the following will make things a bit clearer, the pseudo code was obtained from Wikipedia.

```
;------------------------------------------------------------------------------
; Blatantly stolen from Wikipedia!
; Very slightly modified by Norman Dunbar.
;
; An improved BubbleSort which 'knows' that after each pass, the lowest
; item(s) must be already sorted.
;
; For example:
;
; 9 1 5 3 4, after pass 0, becomes:
; 1 5 3 4 9 so we stop at '4' next time, not at '9'.
;------------------------------------------------------------------------------
```

```
bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
      newn = 0
      for i = 1 to n-1
         Temp = A[i-1]
         if Temp > A[i] then
        A[i-1] = A[i]
        A[i] = Temp
        newn = i
         end if
      end for
      n = newn
    until n = 0
end procedure
;-------------------------------------------------------------------------
```

Listing 2.1: Bubblesort Algorithm

From the above algorithm, we can see that a byte of data will be looked at and using comparisons and swaps, will ‘bubble’ its way to the lower end of the array - that’s the bit furthest from the word count in a QDOSMSQ string, for example.

An example is called for, we start with the test harness which sets up a tiny array of 4 upper case letters, with a leading word count, and sorts it.

```
start
        lea   stuff,a1             ; Where the data are
        bsr.s print_it             ; Print data to #1 unsorted
        bsr.s bubblesort           ; D0.L will be zero
        bsr.s print_it             ; Print sorted data to #1
        rts

stuff   dc.w stuff_end-stuff-2
        dc.b 'C','A','D','B'
stuff_end    equ *
```

Listing 2.2: Bubblesort Test Harness

The code above needs to call a helper routine to print the before and after data, that code follows and is a slightly modified version of the code to find channel #1 and print a string, from the last issue where we were printing the name list.

```
;-------------------------------------------------------------------------
; Some hopefully familiar code from last issue, to print some data
; to channel #1 which MUST BE OPEN.
;-------------------------------------------------------------------------
bv_chbas    equ $30                ; Offset to channel table.

;-------------------------------------------------------------------------
; Find #1 in the channel table. We shouldn't be off the end of the
; table, so NOT CHECKED.
; We assume #1 is open too, so that's NOT CHECKED for either.
;-------------------------------------------------------------------------
print_it
       move.l a1,-(a7)             ; A1 is in use, preserve it

```

```
findChan
        moveq #40,d1              ; Offset to entry #1
        move.l bv_chbas(a6),a0    ; Channel table base offset
        adda.l d1,a0              ; Required entry for #1
        move.l 0(a6,a0.l),a0      ; A0 is ID of channel #1

;------------------------------------------------------------------------------
; Print the text we read from the name list to channel #1.
; Corrupts D1-D3/A1. Preserves A0/A2-A3. D0 = error code.
;------------------------------------------------------------------------------
printText
        move.w ut_mtext,a2        ; Vector to print a string
        jsr (a2)                  ; Print it

;------------------------------------------------------------------------------
; Print a linefeed to channel #1.
; Corrupts D1/A1. Preserves D2-D3/A0/A2-A3. D0 = error code.
;------------------------------------------------------------------------------
linefeed
        moveq #io_sbyte,d0        ; Print a byte trap
        moveq #10,d1              ; Linefeed character
        moveq #-1,d3              ; Timeout
        trap #3                   ; Do it

        move.l     (a7)+,a1       ; Retrieve A1
        rts
```

Listing 2.3: Bubblesort Test Harness

So far so simple, the following is my version of the pseudo code from Wikipedia, converted into assembly language. The labels are named in such a way as, hopefully, to give you an idea of where we are in the pseudo code as converted. Some bits don't convert exactly, the FOR loop, for example, starts with D2=0 and gets incremented by 1 before the loop, not at the end as per a normal FOR loop. But you get the idea, I hope!

The working registers are listed in the comments so that you can, if you wish, follow what's going on.

```
;------------------------------------------------------------------------------
; ENTRY:
;
; A1.L = Start address of bytes to be sorted. Word count first.
;
;------------------------------------------------------------------------------
; WORKING:
;
; A1.L = Start Address of bytes to be sorted, word count first.
; A2.L = Bytes being compared right now. (-1(a2) and (a2)).
; D0.W = 'n' = end of unsorted data.
; D1.B = Temp for swapping.
; D2.W = 'i' = loop counter.
; D3.W = 'newn' = last item sorted.
;------------------------------------------------------------------------------
; EXIT:
;
; D0.L = 0.
; A1.L = Preserved - Start address of sorted bytes' word count.
```

```
; All other registers preserved.
;------------------------------------------------------------------------------
bubblesort
        movem.l d1-d3/a1-a2,-(a7)
        move.w  (a1)+,d0            ; N = length(a)
        beq.s   bs_done
        subq.w  #1,d0               ; We need n-1 when testing

bs_repeat   equ *                   ; Repeat
        movea.l a1,a2               ;   A2 = First unsorted byte
        moveq   #0,d3               ;   Newn = 0

bs_for_loop
        moveq   #0,d2               ;   For i = 1 to n-1

bs_next
        addq.b  #1,d2
        move.b  (a2)+,d1            ;      Temp = A[i-1]
        cmp.b   (a2),d1             ;      If Temp > A[i] then
        bls.s   bs_end_if           ;       Skip swap if A[i-1] <= A[i]

bs_swap
        move.b  (a2),-1(a2)         ;       A[i-1] = A[i]
        move.b  d1,(a2)             ;       A[i] = Temp
        move.w  d2,d3               ;       Newn = i

bs_end_if   equ *                   ;     end if
        cmp.w   d2,d0               ;     I = n-1 yet?
        bne.s   bs_next             ;   End for
        move.w  d3,d0               ;   N = newn
        tst.w   d0                  ;   N = 0 yet?

bs_until
        bne.s   bs_repeat           ; Until n = 0

bs_done
        movem.l (a7)+,d1-d3/a1-a2
        clr.l   d0
        rts
```

Listing 2.4: Bubblesort

So, type the above into a file, save it, assemble it in the usual manner with Gwasl and then load it into a reserved area of memory (mine is 98 bytes long) and simply CALL it. You should see two lines of text on channel #1. The second line being the sorted version of the first.

### 2.0.1 Useful Improvements

The above is fine for sorting the characters in a QDOSMSQ string, and that's the only sorting I actually *need* for my current little project, however, with a couple of minor changes, we can make it even more useful and allow us to sort words, longs and even arrays of strings, if we wish. One way to do this would be to duplicate the code above as many times as we need and edit it accordingly, but that is wasteful even in these days of QPC and other emulators allowing multi-megabytes of RAM. We need a little redesign.

If we extract the compare and swap code to a separate subroutine, we can call it from the main loop, but rather than using a `BSR` instruction, we can use an address register to hold the compare and swap code's address, and use `JSR (An)` instead. That way, we only need to set up the address register once, with the desired compare and swap code's address, and we can reuse most of the above code.

Here's the slightly more useful version of the above code - which can replace the above, from line 51 onwards.

```
51 ;------------------------------------------------------------------------------
52 ; ENTRY:
53 ; For entry at label bubblesort:
54 ;
55 ; A1.L = Start address of data to be sorted. Word count first.
56 ;
57 ;------------------------------------------------------------------------------
58 ; WORKING:
59 ;
60 ; A1.L = Start Address of data to be sorted, word count first.
61 ; A2.L = Data being compared right now. (-1(a2) and (a2)).
62 ; A3.L = Address of the Compare and swap routine.
63 ; D0.W = 'n' = end of unsorted data.
64 ; D1.B = Temp for swapping.
65 ; D2.W = 'i' = loop counter.
66 ; D3.W = 'newn' = last item sorted.
67 ;------------------------------------------------------------------------------
68 ; EXIT:
69 ;
70 ; D0.L = 0.
71 ; A1.L = Preserved - Start address of sorted bytes' word count.
72 ; All other registers preserved.
73 ;------------------------------------------------------------------------------
74 bubblesort
75         movem.l d1-d3/a1-a2,-(a7)
76         move.w (a1)+,d0          ; N = length(a)
77         beq.s bs_done
78         subq.w #1,d0             ; We need n-1 when testing
79 
80 bs_repeat   equ *                ; Repeat
81         movea.l a1,a2            ;   A2 = First unsorted byte
82         moveq #0,d3              ;   Newn = 0
83 
84 bs_for_loop
85         moveq #0,d2              ;   For i = 1 to n-1
86 
87 bs_next
88         addq.b #1,d2
89         jsr (a3)                 ;      Compare and swap if necessary
90 
91 bs_end_if   equ *                ;     end if
92         cmp.w d2,d0              ;     I = n-1 yet?
93         bne.s bs_next            ;   End for
94         move.w d3,d0             ;   N = newn
95         tst.w d0                 ;   N = 0 yet?
96 
97 bs_until
98         bne.s bs_repeat          ; Until n = 0
```

```

bs_done
        movem.l (a7)+,d1-d3/a1-a2
        clr.l d0
        rts
```

Listing 2.5: Better Bubblesort

In the three example compare and swap routines, see Listing 2.6, 2.7 and 2.8, the usage of the working registers is described in Table 2.1.

| Register | Description |
|---|---|
| A1.L | Start Address of data to be sorted. |
| A2.L | Data being compared right now. |
| A3.L | Address of the Compare and swap routine. |
| D0.W | ‘n’ = end of unsorted data. |
| D1.B | Temp for swapping |
| D2.W | ‘i’ = loop counter |
| D3.W | ‘newn’ = last item sorted |

Table 2.1: Working Registers for Bubblesort Compare and Swap Code

```
cas_b
        move.b (a2)+,d1         ; Temp = A[i-1]
        cmp.b (a2),d1           ; If Temp > A[i] then
        bls.s casb_exit         ;   Skip swap if A[i-1] <= A[i]

casb_swap
        move.b (a2),-1(a2)      ; A[i-1] = A[i]
        move.b d1,(a2)          ; A[i] = Temp
        move.w d2,d3            ; Newn = i

casb_exit   rts
```

Listing 2.6: Bubblesort - Compare and Swap - Bytes

The first action required by the code is to grab the current value to be compared. This is pointed to by A2 on entry and is incremented to point at the next entry. In the above, this is byte sized, but see Listing 2.6, 2.7 and 2.8 for subroutines that compare and swap word and long word sized data. The data from the table is loaded into the ‘temp’ variable, also known as D1.size, where size is .B, .W or .L appropriately depending on which compare and swap code we are running.

The comparison between table entries A[i-1] and A[i], from the pseudo code description, actually compares ‘temp’ with ‘A[i]’, or D1.size with (A2), but it’s the same comparison.

In the event that the data in D1 is larger (in this case) than the data in the table pointed to by A2, a swap is made and we set ‘newn’ to the index of the last swap made. We only swap when D1 is larger, that way we don’t end up swapping data that are the same. We are running an inefficient algorithm after all, there’s no need to make it any more inefficient than we have to.

The ‘newn’ variable tells the main loop of the code to stop comparing because whatever index into the table was last swapped, is where the sorted part of the table begins. We don’t need to compare our current value (in D1) with any entries in the table from ‘newn’ onwards.

The following two subroutines can be used to sort arrays of word and/or long words. All that was changed was the size of the data loaded into D1, the `CMP` instruction and the data that are swapped around.

```
cas_w
        move.w (a2)+,d1         ; Temp = A[i-1]
        cmp.w (a2),d1           ; If Temp > A[i] then
        bls.s casw_exit         ;   Skip swap if A[i-1] <= A[i]

casw_swap
        move.w (a2),-2(a2)      ; A[i-1] = A[i]
        move.w d1,(a2)          ; A[i] = Temp
        move.w d2,d3            ; Newn = i

casw_exit    rts
```

Listing 2.7: Bubblesort - Compare and Swap - Words

```
cas_l
        move.l (a2)+,d1         ; Temp = A[i-1]
        cmp.l (a2),d1           ; If Temp > A[i] then
        bls.s casl_exit         ;   Skip swap if A[i-1] <= A[i]

casl_swap
        move.l (a2),-4(a2)      ; A[i-1] = A[i]
        move.l d1,(a2)          ; A[i] = Temp
        move.w d2,d3            ; Newn = i

casl_exit    rts
```

Listing 2.8: Bubblesort - Compare and Swap - Long Words

In our test harness, the code requires to be modified to add a pointer to the desired compare and swap routine in register A3, as follows:

```
start
        lea stuff,a1            ; Where the data are
        lea cas_b,a3            ; Compare and swap bytes
        bsr.s print_it          ; Print data to #1 unsorted
        bsr.s bubblesort        ; D0.L will be zero
        bsr.s print_it          ; Print sorted data to #1
        rts

stuff
        dc.w stuff_end-stuff-2
        dc.b 'C','A','D','B'

stuff_end    equ *
```

Listing 2.9: Bubblesort Test Harness Revisited

If we were sorting an array of word or long word data, we would simply point A3 at the appropriate subroutine, and that's the only difference.

So far, so good, we have the ability to sort bytes, word and long word based data. What about strings? Well, they are a little different and comparing strings is slightly more complicated than a simple `cmp.l (a2),d1` instruction, for example. I'll continue with string sorting in the next issue, for now, we can be satisfied with bytes, words and long words.

There, I think that's all sorted now!

# 3. Printing Multiple Strings at Once

Have you ever needed to print multiple strings, one after the other, perhaps with a linefeed between each one? Neither have I until recently. So if you ever find yourself needing to do exactly that, then the following short utility might be of some help.

```
;------------------------------------------------------------------------------
; MULTIPRINT: Prints numerous strings to the channel in A0.L from a
; table of strings at A1.L. The table format is as follows:
;
; strings   dc.w n                  ; How many strings?
; s1        dc.w s1e-s1-2           ; Size of string 1
;           dc.b '...'              ; Bytes of string 1
; s1e       ds.w 0                  ; Padding byte if required
; s2        dc.w s2e-s2-2           ; Size of string 2
;           dc.b '...'              ; Bytes of string 2
; s2e       ds.w 0                  ; Padding byte if required
;                                   ; And so on.
;------------------------------------------------------------------------------
; REGISTER USAGE:
;
; ENTRY:
;
; A0.L = Channel ID to be used for output.
; A1.L = Start of strings table.
;
; EXIT:
;
; D0.L = Error code or zero. Z flag set accordingly.
; A1.L = Corrupted.
; All other registers preserved.
;------------------------------------------------------------------------------
; ENTRY POINTS:
;
```

```
; MULTIPRINT - Enter here to print the table of strings exactly as is
; with no additional linefeeds etc between strings. If you want any
; linefeeds, you need to define them in the strings.
;
; MULTIPRINT_LF - Enter here to print the strings with a linefeed
; printed after each one. There will be a linefeed at the end, after
; the final string too.
;------------------------------------------------------------------------------
; WORKING REGISTERS:
;
; D7.L = $0A if linefeeds are requested, zero otherwise.
; D6.W = Strings still to print counter.
; A0.L = Channel ID being printed to.
; A1.L = Running pointer to next string to print.
; A2.L = Used to call QDOSMSQ vector to print a string.
; Others - As required by QDOSMSQ vectors and trap calls.
;------------------------------------------------------------------------------

timeout    equ -1                 ; Timeout for TRAP #3 calls
linefeed   equ $0A                ; Linefeed character

;------------------------------------------------------------------------------
; MULTIPRINT_LF.
;------------------------------------------------------------------------------
Multiprint_lf
    move.l d7,-(a7)               ; Save Linefeed indicator
    moveq #linefeed,d7            ; We want linefeeds
    bra.s mp_saveregs             ; And drop in below

;------------------------------------------------------------------------------
; MULTIPRINT.
;------------------------------------------------------------------------------
Multiprint
    move.l d7,-(a7)               ; See main text
    clr.l d7                      ; No linefeeds required

mp_saveregs
    movem.l d1-d3/d6-d7/a2,-(a7)  ; Save working registers + D7 again!
    move.w (a1)+,d6               ; Fetch counter value
    bra.s mp_next                 ; Skip loop first time

mp_loop
    move.l a1,-(a7)               ; Save current string
    move.w ut_mtext,a2            ; Get the vector
    jsr (a2)                      ; Print current string
    bne.s mp_oops                 ; Something bad happened
    move.l (a7)+,a1               ; Start of current string
    adda.w (a1),a1                ; Add size word
    addq.l #3,a1                  ; Prepare to make even
    move.l a1,d5
    bclr #0,d5                    ; D5 now points at next string
    move.l d5,a1                  ; Back into A1

mp_lf
    move.b d7,d1                  ; Linefeed or zero
    beq.s mp_next                 ; Not printing linefeeds
```

```
    moveq #io_sbyte,d0          ; Print a byte
    moveq #timeout,d3
    trap #3                     ; Print linefeed
    tst.l d0
    bne.s mp_done               ; Something bad happened

mp_next
    dbf d6,mp_loop              ; Go around again
    clr.l d0                    ; No errors detected
    bra.s mp_done               ; Clean up on the way out

mp_oops
    adda.l #4,a7                ; Remove saved A1.L

mp_done
    movem.l (a7)+,d1-d3/d6-d7/a2  ; Restore working registers
    move.l (a7)+,d7             ; Restore original D7 again

mp_exit
    tst.l d0                    ; Set the Z flag as necessary
    rts
;------------------------------------------------------------------------------
```

Listing 3.1: Multiprint Utility

### 3.0.2 Stacking D7 Twice? Why?

When I originally wrote this code, I explicitly saved the entry value of register D7, by itself, in `multiprint_lf` but not in `multiprint` where it was the linefeed indicator value that was stacked along with the other working registers. When the code was almost done, it popped the working registers off the stack and checked D7 for zero at `mp_done`. If it was not zero, I popped D7 off the stack again - assuming that we had entered at `multiprint_lf`. Can you see the ever so slightly insidious bug there?

What happens if I enter the code at `multiprint` with D7 already set to zero, when the utility was done, it would pop D7 off the stack, and check it and on finding it to be zero, would attempt to pop another D7 off the stack, assuming that we had entered at `multiprint_lf`. D7 would be loaded with the *calling code's return address* from the stack as opposed to its original value, and so the final `RTS` would cause a crash.

The solution is as per the code above, D7 gets stacked by both utility routines and will always be popped off at the end, twice. That helps keep the stack neat and tidy and avoids this particular intermittent bug/crash.

### 3.0.3 Testing MultiPrint

To test the utility code, all you need is something line the following which I've saved typing time and effort by setting up as yet another filter program which allows me to pass a channel number on the command line, and the output will go to that channel. Lazy? me? ;-)

```
me         equ -1               ; This job
channel_id equ $02              ; Offset(A7) to input file id

start
```

```
        bra   start_2
        dc.l  $00
        dc.w  $4afb

name
        dc.w  name_end-name-2
        dc.b  'MultiPrint Test'

name_end    equ   *

version
        dc.w  vers_end-version-2
        dc.b  'Version 1.00'

vers_end    equ *

str_table
        dc.w 4

s1      dc.w s1e-s1-2
        dc.b 'This is a demo of MultiPrint '
s1e     equ *
        ds.w 0

s2      dc.w s2e-s2-2
        dc.b 'which shows how easy it is to '
s2e     equ *
        ds.w 0

s3      dc.w s3e-s3-2
        dc.b 'print multiple strings in one easy manner. '
s3e     equ *
        ds.w 0

s4      dc.w s4e-s4-2
        dc.b 'Written by Norman Dunbar',$0a
s4e     equ *
        ds.w 0


start_2
        move.l channel_id(a7),a0   ; channel id
        lea   str_table,a1        ; Table of strings
        bsr   MultiPrint          ; Print with no linefeeds

        lea   str_table,a1        ; Table of strings again
        bsr   MultiPrint_lf       ; Print with linefeeds between

        moveq #0,d3               ; No error code
        moveq #mt_frjob,d0
        moveq #me,d1              ; This job is about to die
        trap  #1

        in "ram1_MultiPrint_lib"
```

Listing 3.2: Testing the Multiprint Utility

And finally, the ram1_MultiPrint_lib file will look like this. However, if you have changed the code layout above (for MultiPrint_asm) then you may have to regenerate the lib file using the SYM_bin utility.

```
MULTIPRINT_LF    EQU      *+$00000000
MULTIPRINT       EQU      *+$00000006

                 lib "ram1_multiprint_bin"
```

Listing 3.3: The Multiprint Library File

You should execute the test harness as follows:

```
ex ram1_MultiPrint_test_bin , #1
```

Listing 3.4: Executing the Multiprint Test Harness

And the output will be something like the following:

```
This is a demo of MultiPrint which shows how easy it is to print
multiple strings in one easy manner. Written by Norman Dunbar
This is a demo of MultiPrint
which shows how easy it is to
print multiple strings in one easy manner.
Written by Norman Dunbar
```

Listing 3.5: Results of the Multiprint Test Harness

The first couple of lines shows the data printed “as is” without linefeeds. The remainder of the output shows each string printed with a separating linefeed.

Because I had my channel #1 defined as a quite narrow window, the first line of output wrapped around onto the next line, in the normal manner of printing long strings.

Because there are now two linefeeds after the final string, we get a blank line after the final one. Or, we will when the next print to that channel takes place, it’s possible that QDOSMSQ has the final linefeed as pending. I noticed that in testing occasionally.

# 4. Hexdump Utility

I'm a frequent user of the Linux/Unix `hexdump` utility in my real life, and I miss it on QDOSMSQ. I decided to put that right and as a continuation of the use of filter utilities in a previous issue, I decided to make this utility a filter too.

To execute the utility, you simply:

```
ex win1_hexdump_bin, source_file, dest_location
```

Listing 4.1: Executing the Hexdump Utility

The source file should be obvious, it's the one you want to examine, and the dest_location can be either a filename or a channel number.

So, without any further ado, here's the code. I'll explain it at the end, but it's fairly simple.

### 4.0.4 Hexdump Listing

```
;---------------------------------------------------------------------------
; HEXDUMP:
;
; A filter program using an input and output channel, passed on
; the stack for it's files.
;
; EX hexdump_bin, binary_file, output_file
;
;---------------------------------------------------------------------------
; 21/09/2015 NDunbar Created for QDOSMSQ Assembly Mailing List
;---------------------------------------------------------------------------
; (c) Norman Dunbar, 2015. Permission granted for unlimited use
; or abuse, without attribution being required. Just enjoy!
;---------------------------------------------------------------------------

me          equ -1                  ; This job
```

```
infinite    equ -1                 ; For timeouts
err_bp      equ -15                ; Bad parameter error
linefeed    equ $0A                ; Linefeed character
eof         equ -10                ; End of file
buff_size   equ $10                ; Maximum size of read buffer
out_size    equ 73                 ; Output string length
space       equ ' '                ; 1 space
dot         equ '.'                ; 1 dot
max_char    equ $C0                ; Highest printable ASCII character

source_id   equ $02                ; Offset(A7) to input file id
dest_id     equ $06                ; Offset(A7) to output file id
param_size  equ $0A                ; Offset(A7) to command string size
param       equ $0C                ; Offset(A7) to command bytes

start
        bra  Hexdump
        dc.l $00
        dc.w $4afb

name
        dc.w name_end-name-2
        dc.b 'Hexdump'

name_end    equ  *

version
        dc.w vers_end-version-2
        dc.b 'Version 1.00'

vers_end    equ *

in_buffer
        ds.l 4                 ; 16 bytes read at a time

out_buffer
        ds.l 20                ; 80 bytes max output

open_bracket equ out_buffer+54     ; Where '[' should be
close_bracket equ out_buffer+71    ; Where ']' should be

;-------------------------------------------------------------------------
; Stack on entry:
;
; $0c(a7) = bytes of parameter + padding, if odd length. (Ignored)
; $0a(a7) = Parameter size word. (Ignored)
; $06(a7) = Output file channel id.
; $02(a7) = Source file channel id.
; $00(a7) = How many channels? Should be $02.
;-------------------------------------------------------------------------
bad_parameter
        moveq #err_bp,d0       ; Guess!
        bra suicide            ; Die horribly

Hexdump
        cmpi.w #$02,(a7)       ; Two channels is a must
```

```
        bne.s bad_parameter     ; Oops

start_loop
        moveq #infinite,d3      ; Timeout - preserved throughout
        clr.l d7                ; Current location in file

read_loop
        move.l source_id(a7),a0  ; Input channel id
        lea in_buffer,a1          ; Where to read the data into
        moveq #buff_size,d2     ; Maximum size of the buffer
        moveq #io_fstrg,d0      ; Trap utility we want
        trap #3                 ; Read a chunk of source file
        tst.l d0                ; Did it work?
        beq.s read_ok           ; Not EOF yet, carry on
        cmpi.l #eof,d0          ; EOF?
        bne error_exit          ; Something bad happened
        tst.w d1                ; Any remaining data?
        beq all_done            ; No, exit the main loop

read_ok
        lea in_buffer,a2        ; Source buffer
        lea out_buffer,a1       ; Output buffer
        moveq #79,d0            ; 80 bytes to clear

;------------------------------------------------------------------
; Space fill the entire output buffer on each pass through the loop.
;------------------------------------------------------------------
ob_clear
        move.b #space,(a1,d0.w) ; Space fill from the end back
        dbf d0,ob_clear         ; And do the rest
        moveq #0,d5             ; Extra linefeed counter

;------------------------------------------------------------------
; Add the address to the buffer as 8 hex characters. Then 4 spaces.
;------------------------------------------------------------------
hd_address
        move.l d7,d4            ; D4 is required here
        beq.s hd_continue       ; No extra linefeed at start
        cmpi.b #0,d7            ; On a 256 Byte boundary?
        bne.s hd_continue       ; Nope.
        move.b #linefeed,(a1)+  ; Yes, extra linefeed
        moveq #1,d5             ; Adjust counter

hd_continue
        ext.l d1                ; Curently only word sized
        add.l d1,d7             ; Update file offset counter
        bsr hex_l               ; Store address in buffer at A1
        adda.l #4,a1            ; Leave 4 spaces

;------------------------------------------------------------------
; There might not always be 16 bytes to convert. Adjust the count to
; add groups of 4 bytes then two spaces to the output buffer, by
; counting long words and then the remaining spare bytes.
;------------------------------------------------------------------
hd_data
        move.l d1,d0            ; Byte counter (long sized)
```

```
        divu #4,d0              ; D0.Low = Long word count
;                               ; D0.High = Byte count remainder
        bra.s hdl_next          ; Skip first time

hdl_loop
        move.l (a2)+,d4         ; Get a long word
        bsr.s hex_l             ; Add hex to buffer
        adda.l #2,a1            ; Leave 2 spaces between groups

hdl_next
        dbf d0,hdl_loop         ; Do next long word

        swap d0                 ; D0.W = remaining bytes (0-3)
        bra.s hdb_next          ; Skip first byte

hdb_loop
        move.b (a2)+,d4         ; Get a byte
        bsr.s hex_b             ; Add to buffer

hdb_next
        dbf d0,hdb_loop         ; Do next byte

;------------------------------------------------------------------------------
; Because we don't always get 16 bytes, we simply force A1 to the
; desired location in the output buffer.
;------------------------------------------------------------------------------
hd_ascii
        lea open_bracket,a1     ; where to put the '['
        adda.w d5,a1            ; Adjust for extra linefeeds
        lea in_buffer,a2        ; Back to the start of data
        move.w d1,d0            ; Data counter
        move.b #'[',(a1)+       ; Opening delimiter added

        bra hda_next            ; Skip first time

hda_loop
        move.b (a2)+,d2         ; Fetch byte of data
        cmpi.b #space,d2        ; We can print space or higher only
        bcs.s hda_dot           ; This character is not ok
        cmpi.b #max_char,d2     ; Reached the control characters?
        bcs.s hda_store         ; No, this one is fine

hda_dot
        moveq #dot,d2           ; Print a dot instead

hda_store
        move.b d2,(a1)+         ; Save in output buffer

hda_next
        dbf d0,hda_loop         ; And do the rest

        lea close_bracket,a1    ; Where to put the ']'
        adda.w d5,a1            ; Adjust for extra linefeeds
        move.b #']',(a1)+       ; Closing delimiter added
        move.b #linefeed,(a1)   ; And linefeed at the end

```

```
hd_print
        moveq #io_sstrg,d0       ; Trap call we want
        moveq #out_size,d2       ; How many bytes?
        add.w d5,d2              ; Adjust for extra linefeeds
        lea out_buffer,a1        ; Where our string is
        move.l dest_id(a7),a0    ; Output channel
        trap #3                  ; Do it
        tst.l d0                 ; Did it work?
        beq read_loop            ; Yes, continue

error_exit
        move.l d0,d3             ; Error code we want to return
        bra.s suicide            ; And die

all_done
        moveq #0,d3              ; No error code

suicide
        moveq #mt_frjob,d0
        moveq #me,d1             ; This job is about to die
        trap #1

;-----------------------------------------------------------------------
; The hex conversion routines in QDOS are corrupt in some versions so
; these will work. The take a long, word, byte or nibble in D4 and
; write the hex byte(s) to a buffer pointed to by A1.
;
; The various routines here call a lower level one, then drop into
; the called code again to process the "other half" of the data to be
; converted.
;-----------------------------------------------------------------------
hex_l
        swap d4                  ; We do this in MS word order
        bsr.s hex_w              ; Do original high word
        swap d4                  ; Get low word back

hex_w
        ror.w #8,d4              ; We do this in MS byte order
        bsr.s hex_b              ; Do original high byte
        rol.w #8,d4              ; Get low byte back

hex_b
        ror.b #4,d4              ; We do this in MS nibble order
        bsr.s hex_nibble         ; Do original high nibble
        rol.b #4,d4              ; Get original low niggle back

hex_nibble
        move.b d4,-(a7)          ; We need to save the byte
        andi.b #$0f,d4           ; Mask out low nibble
        addi.b #'0',d4           ; Assume digit 0-9
        cmpi.b #'9',d4           ; Digit?
        bls.s hex_store          ; Yes, digit
        addi.b #7,d4             ; Offset for an A-F character

hex_store
        move.b d4,(a1)+          ; Add to the buffer at A1.L
```

```
        move.b  (a7)+,d4            ; Retrieve original byte
        rts
```

Listing 4.2: Hexdump Utility

### 4.0.5 Hexdump Code Explained

As ever with my code, the first part is a load of bumff explaining briefly, sometimes, what the program should be doing. This utility is no different! Following on, we have a number of equates defined. The important ones here should be adequately commented - but we set up various offsets onto the A7 stack to extract the source file and destination channel ids and, not *currently* used here, where we should find the command string, if passed.

Then there is the usual standard QDOS header for a job with the job name embedded and a couple of buffers. The input buffer is where we read the source file into, 16 bytes at a time. The output buffer is big enough to hold a printed output line of up to 80 characters. You may note that a program version has been defined, but is only for my own documentation, it is never display or used. Feel free to leave it out.

The next couple of equates define the locations in the output buffer where the '[' and ']' surrounding the ASCII representation of the hex codes will be.

Just before the main Hexdump code itself, we have the `bad_parameter` code which is, as you might expect, used to handle bad parameters - these are when we get less than or more than two channels on the stack at execution time. The utility simply exits with an error code back to the caller.

Be aware that you will not see this error code if you `EX` the utility, only if you call it with `EW` will errors be reported back to SuperBasic. This is normal.

`Hexdump` starts by checking the word on the stack to ensure that we only received two channel ids on the stack. If this is not the case, we exit via `bad_parameter` as explained above. Assuming this is not the case, we preload D3 with an infinite timeout. This is preserved through all trap calls, so only needs to be done once.

We use D7 as the current offset counter, so we initialise it to zero, as we are still at the start of the source file.

`Read_loop` is the start of the main loop. In here, we load the source file's channel id into A0 and read the next 16 bytes, maximum, into the input buffer. When we hit end of file, we need to ensure that the last few remaining bytes are converted to hex - if there was not exactly 16 bytes read when we hit EOF, they are still valid. We test D1 to be sure that we do have some data to process, if not, we are truly at EOF and we bale out of the utility passing a zero error code back to the caller.

If there was some other error in the read, ie, not EOF, then we simply bale out and return the error code to the caller.

Assuming all went well, we enter the code at `read_ok` where we set up A2 and A1 with the input and output buffer addresses respectively. As we want spaces in between each section of data in the output buffer, we fill all 80 bytes with spaces, prior to each conversion, at `ob_clear`. D5 is cleared here as well, on each pass, as it counts the number of extra linefeeds that have been injected into the output buffer - zero or one - and is used to adjust various pointers and counts as necessary.

The code at `hd_address` copies the current offset from D7 into D4 and if this is the start of the file - the offset is zero - skips over the next bit. Assuming that this is not the start of the file, we wish to insert an extra linefeed after every 256 bytes of the input file. This is easy to accomplish as we simply need to check the lowest byte of the offset. If it is zero, then we add a linefeed to the buffer and set D5 to 1 to show the extra byte. This happens at offsets $0100, $0200, $0300 and so on.

Prior to updating D7 with the count of the bytes just read. For most of the file, this will be 16 but there may be less at EOF. As the offset in D7 is long sized - we could be dumping large files - we have to extent D1 from a word to a long prior to the addition. D4 is converted from an offset to 8 hex characters in a call to `hex_l` which adds the converted characters to the output buffer and updates A1.

After the address has been added, we wish to have 4 spaces after it, so A1 is incremented by 4 to account for this. We are now ready to convert the data.

`Hd_data` is where this happens. The bytes read is copied to D0 as a long word and then divided by 4 to get the number of long words read in. In most cases this will be 4, at least until we get to EOF. After the division, the low word of D0 holds the number of long words to convert and the high word holds the remaining bytes to convert afterwards. Each long word is converted by copying it to D4.L and calling out to the `hex_l` code again to convert and add it to the buffer as 8 hex characters. Two spaces are then 'added' by incrementing A1 accordingly.

After all the long words are converted, we process the remaining bytes by swapping D0 around so that the remaining bytes are in the low word, and we loop around those converting them one byte at a time at `hdb_loop`.

After all the bytes are processed and added to the buffer, we need to add in the ASCII characters. Only printable ones will be considered - those between 'space' and the down arrow character, inclusive. Anything less than a space or any of the control characters from $C0 upwards are represented by a dot.

The first part of the code at `hd_ascii` adds an opening bracket to the buffer, then the individual ASCII characters are added, all 16 (usually) of them, then a closing bracket is added to the buffer followed by a linefeed. If we injected an extra linefeed previously, then D5 is added to the offsets for the opening and closing brackets to ensure that they are inserted into the buffer at the correct location.

We then drop into `hd_print` where we send the completed buffer, to the destination file or channel before looping around and back to `read_loop` to do it all again. Once again, the counter in D2 which determines the size of the string to print has to be adjusted to account for any extra linefeeds, so D5 is added to D2 before the `TRAP #3`.

In the unlikely event of an error during the conversion to hex, the code at `error_exit` will be executed to copy the error code from D0 into D3 prior to returning to the caller. If there were no errors, then `all_done` will cause a zero to be returned. The job then kills itself which will cleanly close the input and output files, flushing any buffers as appropriate.

### 4.0.6 Hex Conversion

As noted in the comments, certain versions of QDOS, prior to 1.03 I believe, have hex conversion routines in the ROM, but they are somewhat broken. To this end, I have supplied my own. To use them, D4 should contain the value to be converted and A1 should point to a location in a buffer, somewhere, for the results. After conversion, A1 is updated to the next free location in the buffer.

The following is a sample of the output from the utility when used to hexdump an earlier incarnation[1] of itself.

[1] A *much* earlier version!

```
00000000  60000078 00000000 4AFB0007 48657864  [`..x....J...Hexd]
00000010  756D7000 61736D00 00000000 00000000  [ump.asm.........]
00000020  00000000 00000000 00000000 00000000  [................]
00000030  66EDE055 00010002 00000000 00000000  [f..U............]
00000040  00000000 00000000 00000000 00000000  [................]
00000050  00000000 00000000 00000000 00000000  [................]
00000060  00000000 00000000 00000000 00000000  [................]
00000070  00000000 70F16000 00C00C57 000266F4  [....p.`....W..f.]
00000080  76FF4287 206F0002 43FAFF8A 74107003  [v.B. o..C...t.p.]
00000090  4E434A80 67100C80 FFFFFFF6 66000094  [NCJ.g.......f...]
000000A0  4A416700 009245FA FF6C43FA FF78704F  [JAg...E..lC..xpO]
000000B0  13BC0020 000051C8 FFF82807 48C1DE81  [...  ..Q...(.H...]
000000C0  617CD3FC 00000004 200180FC 0004600A  [a|......  .....`.]
000000D0  281A616A D3FC0000 000251C8 FFF44840  [(.aj......Q...H@]
000000E0  6004181A 616451C8 FFFA43FA FF6E45FA  [`...adQ...C..nE.]
000000F0  FF243001 12FC005B 60000014 141A0C02  [.$0....[`.......]

00000100  00206506 0C0200C0 6502742E 12C251C8  [. e.....e.t...Q.]
00000110  FFEC43FA FF5712FC 005D12BC 000A7007  [..C..W...]....p.]
00000120  744943FA FF00206F 00064E43 4A806700  [tIC... o..NCJ.g.]
00000130  FF542600 60027600 700572FF 4E414844  [.T&.`.v.p.r.NAHD]
00000140  61024844 E05C6102 E15CE81C 6102E91C  [a.HD.\a..\..a...]
00000150  1F040204 000F0604 00300C04 00396304  [.........0...9c.]
00000160  06040007 12C4181F 4E75               [........Nu      ]
```

Listing 4.3: Example Hexdump Output

# 5. Jump Tables

Imagine that your next great programming wonder is not based on the Pointer Environment, but does display a menu to the user with a number of options[1]. Each option can be selected by a single key press, and your application code has to choose a piece of code, a subroutine, to handle the user's choice.

You could do something like the following, where we assume that only the 10 digits are allowed and that D0.B holds the keypress character from the menu.

```
;------------------------------------------------------------------------------
main_loop
        bsr  display_menu        ; CLS and display the menu
        bsr  get_menu_option     ; Wait for a menu choice

got_menu_option
        cmpi.b #'0',d0           ; Zero or above?
        bcs  bad_option          ; Oops
        cmpi.b #'9',d0           ; Nine or below?
        bcc  bad_option          ; Oops

got_good_option
        cmpi.b #'0',d0
        beq  option_0            ; Process option '0'
        cmpi.b #'1',d0
        beq  option_1            ; Process option '1'
        ...
        ...
        cmpi.b #'8',d0
        beq  option_8            ; Process option '8'
        cmpi.b #'9',d0           ; Not strictly required, but safe
        beq  option_9            ; Process option '9'

```

[1] It wouldn't be much of a menu otherwise, would it? :-)

```
option_return
        ; do some post routine clean up here
        ...
        ...
        bra main_loop           ; Ready for the next option

option_0
        ; Process option zero here.
        ...
        bra option_return       ; Back to the main loop

option_1
        ; Process option one here.
        ...
        bra option_return       ; Back to the main loop
        ...
        ...
;------------------------------------------------------------------------------
```

Listing 5.1: Processing User Options - First Attempt

Ignoring the fact that there are numerous helper routines called, but not shown in the above example, then we can see that the above is quite simple to read and is fine for a small number of options. However, note that none of the option handling subroutines can use an `RTS` instruction to exit, as the call to the subroutine was by way of a `BEQ` instruction. They must therefore execute a `bra option_return` to get back into the clean up code and back to the main loop.

We could improve matters slightly and use the `PEA` here to set up a pseudo subroutine call, by pushing the `common_return` address onto the stack prior to calling any of the subroutines, as follows.

```
;------------------------------------------------------------------------------
main_loop
        bsr display_menu        ; CLS and display the menu
        bsr get_menu_option     ; Wait for a menu choice

got_menu_option
        cmpi.b #'0',d0          ; Zero or above?
        bcs bad_option          ; Oops
        cmpi.b #'9',d0          ; Nine or below?
        bcc bad_option          ; Oops

got_good_option
        pea option_return       ; Stack a "return" address

        cmpi.b #'0',d0
        beq option_0            ; Process option '0'
        ...
        ...
        cmpi.b #'9',d0          ; Not strictly required, but safe
        beq option_9            ; Process option '9'

option_return
        ; do some post routine clean up here
        ...
        ...
        bra main_loop           ; Ready for the next option
```

```

option_0
        ; Process option zero here.
        ...
        rts                     ; Back to option_return

option_1
        ; Process option one here.
        ...
        rts                     ; Back to option_return

        ...
        ...
;------------------------------------------------------------------------------
```

Listing 5.2: Processing User Options - Improved First Attempt

This version is a lot better, while we are still calling the subroutines with a `BEQ` instruction, we have fiddled the stack by pushing a common return address onto it when we know we have a valid menu option. When each individual subroutine executes the `RTS` at the end, it will pop the address of `option_return` and continue executing from there.

We could, if we wished to use the actual `BSR` instruction, perhaps to avoid confusion, code something like the following.

```
;------------------------------------------------------------------------------
main_loop
        bsr display_menu        ; CLS and display the menu
        bsr get_menu_option     ; Wait for a menu choice

got_menu_option
        cmpi.b #'0',d0          ; Zero or above?
        bcs bad_option          ; Oops
        cmpi.b #'9',d0          ; Nine or below?
        bcc bad_option          ; Oops

got_good_option
        cmpi.b #'0',d0
        bne.s ggo_try_1         ; Not zero
        bsr option_0            ; Process option '0'
        bra option_return       ; Do cleanup

ggo_try_1
        cmpi.b #'1',d0
        bne.s ggo_try_2         ; Not '1'
        bsr option_1            ; Process option '1'
        bra option_return       ; Do cleanup

        ...
        ...
ggo_try_8
        cmpi.b #'8',d0
        bne.s ggo_try_9         ; Not '8'
        bsr option_8            ; Process option '8'
        bra option_return       ; Do cleanup

ggo_try_9
```

```
        cmpi.b #'9',d0          ; Not strictly required, but safe
        bne.s option_return     ; Not '9'
        bsr option_9            ; Process option '9'
        bra option_return       ; Do cleanup

option_return
        ; do some post routine clean up here
        ...
        ...
        bra main_loop           ; Ready for the next option

option_0
        ; Process option zero here.
        ...
        rts

option_1
        ; Process option one here.
        ...
        rts

        ...
        ...
;------------------------------------------------------------------------------
```

Listing 5.3: Processing User Options - Another Improved First Attempt

So, in this version, we are using the `BSR` instruction that we wanted to, but now we've had to invert all the flag checks after the `cmpi.b #whatever,d0` and add in numerous new labels and branches, plus, after a successful return from the subroutine, we need an explicit branch to the clean up code at the bottom of the loop. It's all getting rather messy now.

You can imagine that as we add more and more menu options, that adding in new subroutines etc could get a bit frantic, especially trying to remember to do all the branches etc. In addition, there's much more typing, and, if you type like I do, too much room for errors![2]

Jump tables are easily set up, and can make life so much easier, with a lot less typing, although, it could be said that they are slightly less easily understood[3].

```
;------------------------------------------------------------------------------
JumpTable
        dc.w option_0-JumpTable
        dc.w option_1-JumpTable
        dc.w option_2-JumpTable
        dc.w option_3-JumpTable
        dc.w option_4-JumpTable
        dc.w option_5-JumpTable
        dc.w option_6-JumpTable
        dc.w option_7-JumpTable
        dc.w option_8-JumpTable
        dc.w option_9-JumpTable

main_loop
```

[2] I've been in the IT business since around 1982, I *still* cannot touch type, I have to look at the keyboard to see where the next key I want is hiding!

[3] At least until you begin to understand exactly how useful they really are!

```
        bsr  display_menu        ; CLS and display the menu
        bsr  get_menu_option     ; Wait for a menu choice

got_menu_option
        cmpi.b #'0',d0           ; Zero or above?
        bcs  bad_option          ; Oops
        cmpi.b #'9',d0           ; Nine or below?
        bcc  bad_option          ; Oops

got_good_option
        subq.b #'0',d0           ; D0.B = 0 to 9 as a number
        ext.w d0                 ; Now extend to a word
        lsl.w #1,d0              ; Convert to a table offset
        lea  JumpTable,a2        ; Where the jump table lives
        jsr  (a2,d0.w)           ; Jump to the correct subroutine

option_return
        ; do some post routine clean up here
        ...
        ...
        bra  main_loop           ; Ready for the next option

option_0
        ; Process option zero here.
        ...
        rts

option_1
        ; Process option one here.
        ...
        rts

        ...
        ...
;------------------------------------------------------------------------------
```

Listing 5.4: Processing User Options - Jump Tables

Each entry in the table surprisingly names `JumpTable` is a word sized *signed* offset to the desired routine, from the start of the table itself. This allows for subroutines that are located prior to, or after, the jump table being defined. Negative offsets are to subroutines defined before the table, and positive offsets are to subroutines defined after the jump table. Simple?

You can see how much less code there is at the label `got_good_option`. At that point all we have to do is convert D0.B from a byte, containing one of the characters '0' through '9', into a word containing the numeric value zero to nine, as opposed to the character '0' to '9', then double it as each entry in the table takes two bytes. The offset to the `option_0` subroutine is at `JumpTable` + 0, while that for the `option_1` subroutine is at `JumpTable` + 2 and so on.

Obviously, the code at `main_loop` is executed without passing through the preceding jump table, or who knows what might happen! Jump tables are data, not code.

The `jsr (a2,d0.w)` takes care of calling the correct routine, as A2 is pre-loaded with the address of `JumpTable`. On return, we drop into the clean up code and pass back to the main loop start again. Remember, D0.W will be sign extended to a long word prior to adding it to A2.L.

Adding new options is a simple matter of inserting or appending a new entry to the jump table *in the correct place*, and making sure that D0.W is set equal to the offset in the jump table, so that when we execute the `jsr (a2,d0.w)` instruction, we get the correct subroutine address.

### 5.0.7 What About Missing Options

So far so good, our table holds one subroutine offset for each menu option from '0' to '9', which gets translated to a value between 0 and 9, and subsequently, into an offset into the table of offset words[4]. What do we do if, for example, option 5 is not actually allowed? We have a couple of choices:

- Filter out the illegal option(s) when checking for a valid choice.
- Use a 'do nothing' entry for the invalid choice(s) in the table.
- Use a zero offset in the table,test for it in the and don't jump if that is found.

The first option is obviously the best as it gives you the opportunity to advise the user of their error when they try to make an invalid choice. The last option would require a slight change to the code at `got_good_option`, as follows:

```
got_good_option
        subq.b #'0',d0          ; D0.B = 0 to 9 as a number
        ext.w d0                ; Now extend to a word
        lsl.w #1,d0             ; Convert to a table offset
        lea JumpTable,a2        ; Where the jump table lives
        tst.w (a2,d0.w)         ; Valid offset?
        beq.s no_jump           ; No, do nothing
        jsr (a2,d0.w)           ; Jump to the correct subroutine
```

Listing 5.5: Processing User Options - Jump Tables

The code at label `no_jump` would do whatever is required prior to the next pass through the main loop.

[4] Ugh! Too many offsets in that sentence!

# 6. Using the MC68020 Instructions

As you may be aware, in all of the articles I published in **QL Today** over the years, and in the preceding issues of this randomly occurring eMagazine, I've been a loyal user of George's Gwasl assembler. This worked well on old black box QLs but who is using one of those these days? Anyone?

It is time to move on from the toys and playthings of childhood and become a real [wo]man. From the next issue, issue 4, we are going to switch to George's other assembler, Gwass and get down and dirty using the 68020 instructions. If you are using QPC, then you are already able to use them as George had a hand in getting QPC running on an emulated 68020 rather than a simple 68008 as the old Black Boxes used to run.

How many of my readers will this upset I wonder? Table 6.1 gives details of which computer or emulator can handle the new instructions.

| **Computer** | **Processor** | **Comments** |
|---|---|---|
| QL | 68008 | Cannot use the new instructions. |
| QPC | 68020 | Able to use the new instructions. |
| Others | 68008 | Cannot use the new instructions |

Table 6.1: Emulators and the 68020

This is a problem perhaps? Does anyone not use QPC for their main "QL on a PC"? Would some or all of my readers be missing out if I went down this route?

You better let me know, soon(ish) at the usual email address `assembly@qdosmsq.dunbar-it.co.uk`.